Gradient Descent Strategies

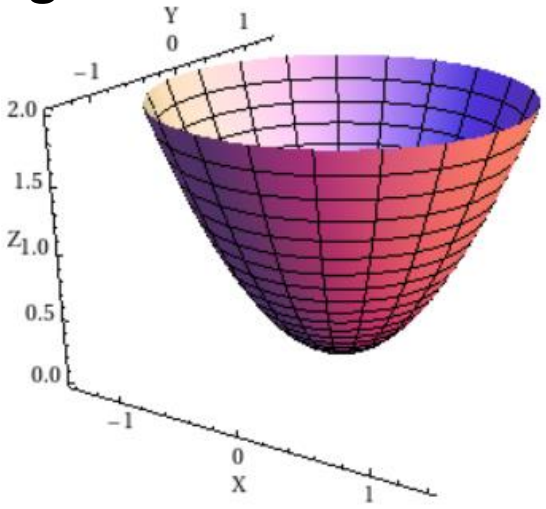
AW

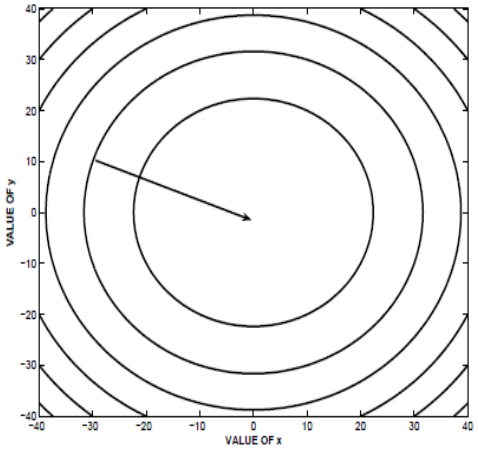
Lecture Overview

1. Tuning Learning Rate
2. Momentum Methods
3. AdaGrad and RMSProp
4. AdaDelta and Adam

Example: Bowllike Loss

- Simplest convex, quadratic low of bowl-like shape and a single global minimum





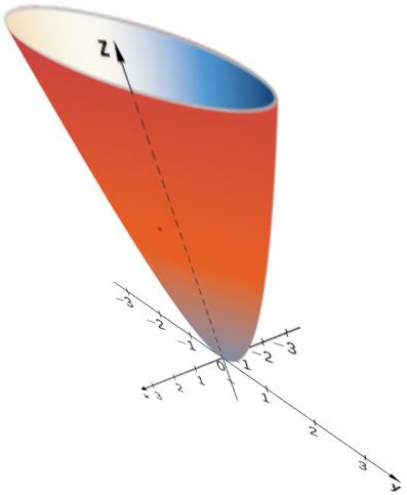
Loss function is circular bowl

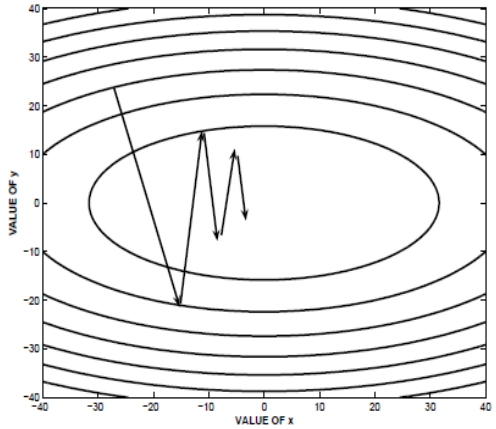
$$L = x^2 + y^2$$

- Symmetric – sensitivity to gradient in both arguments is the same
- The gradient points directly at the optimum solution, and one can reach the optimum in a single step, as long as the correct step-size is used.

Example: Bowllike Loss (cont.)

- Simplest convex, quadratic low of bowl-like shape and a single global minimum





Loss function is elliptical bowl

$$L = x^2 + 4y^2$$

- The gradients are often more significant in the *y*-direction compared to the *x*-direction.
- Steps along in *y*-direction are large, but subsequent steps undo the effect of previous steps. Steps in the *x*-direction is consistent but tiny.
- The gradient never points to the optimal solution, because of which many course corrections are needed over the descent.

Stochastic Gradient Descent and its Problems

- In tf.keras these are all SGD optimizers
- Path of steepest descent in most loss functions is only an instantaneous direction of best movement, and is not the correct direction of descent in the longer term.
- A steepest-descent direction can sometimes become an ascent direction after a small update in parameters
- *The only way to reach the optimum with steepest-descent updates is by using a large number of tiny updates and course corrections*
- The problem of oscillation and zigzagging is quite ubiquitous whenever the steepest-descent direction moves along a direction of high curvature in the loss function

Modifications of SGD

- SGD can be set up as:

```
opt = tf.keras.optimizers.SGD(learning_rate=xxx)
model.compile(loss='mse', optimizer=opt))
model.fit(x, y, nb_epoch=num_epochs, batch_size=yyy, verbose=0, \
                    validation_split=0.1)
```

where xxx and yyy are respectively desire learning rate and bacth size

Depending on how we set up yyy we get

1. yyy=1: Simple SGD when update is done after each sample
2. yyy= all data: Batch GD all the training data is treated as a single step:
   - Take the average of the gradients of all the training examples and then use that mean gradient to update our parameters. So that's just one step of gradient descent in one epoch
3. yyy= portion of the data: Mini-Batch GD:
   - Take the average of the gradients of the training examples in the minibatch and then use that mean gradient over mini-batch to update our parameters, i.e. do one step of gradient descent after each mini-batch of data

Which SGD to use

- Batch GD for smoother loss curves on smooth manifolds. Batch GD converges directly to minima, but of course it can easily overshoot, so it is going to be zigzagging. So you need many epochs – slow, especially on large data-sets.
- SGD does updates on every step so it is going to e zigzagging even more. But it converges faster than BGD on large dataset because we do not need to complete the whole epoch. Yet, since in SGD we use only one example at a time, we cannot implement the vectorized implementation on it. This slows down the computations.
- Mini-batch is a tradeoff between the options: less zigzagging because mini-batch smoothens the gradient, and vectorized computations. But not as long as running full batch

Mitigating Zig-Zagging

- Constant learning rate is a bad idea because of the dilemma :
  - A lower learning rate used early on will cause the algorithm to take too long to come even close to an optimal solution.
  - A large initial learning rate will allow the algorithm to come reasonably close to a good solution at first; but the algorithm will then oscillate around the point for a very long time, or diverge in an unstable way, if the high rate of learning is maintained in vicinity of optimum
- Possible solution: to allow the learning rate to decay over time
- The most common decay functions *exponential decay* and *inverse decay*:

$$\alpha_t = \alpha_0 e^{-kt} \text{ (exponential decay)}$$

$$\alpha_t = \frac{\alpha_0}{1+kt} \text{ (inverse decay)}$$

where α_0 is initial decay, t epoch number, k-controls rate, set up heuristically – often to ½.

- Better momentum methods

Adjusting Learning Rate

- To set up changing learning rate we can do it as follows in Keras:

```
lr_schedule = keras.optimizers.schedules.ExponentialDecay( \
        initial_learning_rate=1e-2, \
        decay_steps=10000, \
        decay_rate=0.9)
opt = keras.optimizers.SGD(learning_rate=lr_schedule)
```

- To implement your own schedule object, you should implement the __call__ method, which takes a step argument (scalar integer tensor, the current training step count).
- Like for any other Keras object, you can also optionally make your object serializable by implementing the get_config and from_config methods.

Lecture Overview

1. Tuning Learning Rate
2. Momentum Methods
3. AdaGrad and RMSProp
4. AdaDelta and Adam

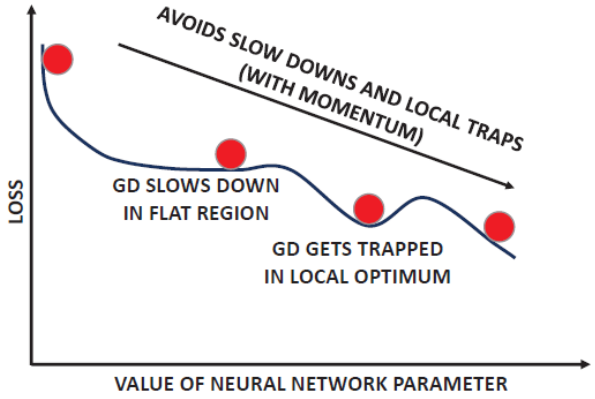
Momentum Methods

- Setting: performing gradient descent with respect to the parameter vector $\vec{w}$
- Normal gradient descent: updates wrt loss L (defined over a mini-batch of instances) are $\vec{w} = \vec{w} + \vec{v}$ where $\vec{v} = -\alpha\left(\frac{\partial L}{\partial \vec{w}}\right)$ and α is a learning parameter
- Momentum methods: updates wrt loss L are $\vec{w} = \vec{w} + \vec{v}$ where $\vec{v} = \beta\vec{v} - \alpha\left(\frac{\partial L}{\partial \vec{w}}\right)$ where $\beta \in \{0,1\}$ is a (*momentum* or *friction*) parameter of *exponential smoothing* of update
  - $\beta = 0$ is straightforward mini-batch gradient-descent
  - Large β helps pick up a consistent velocity $\vec{v}$ in the correct direction

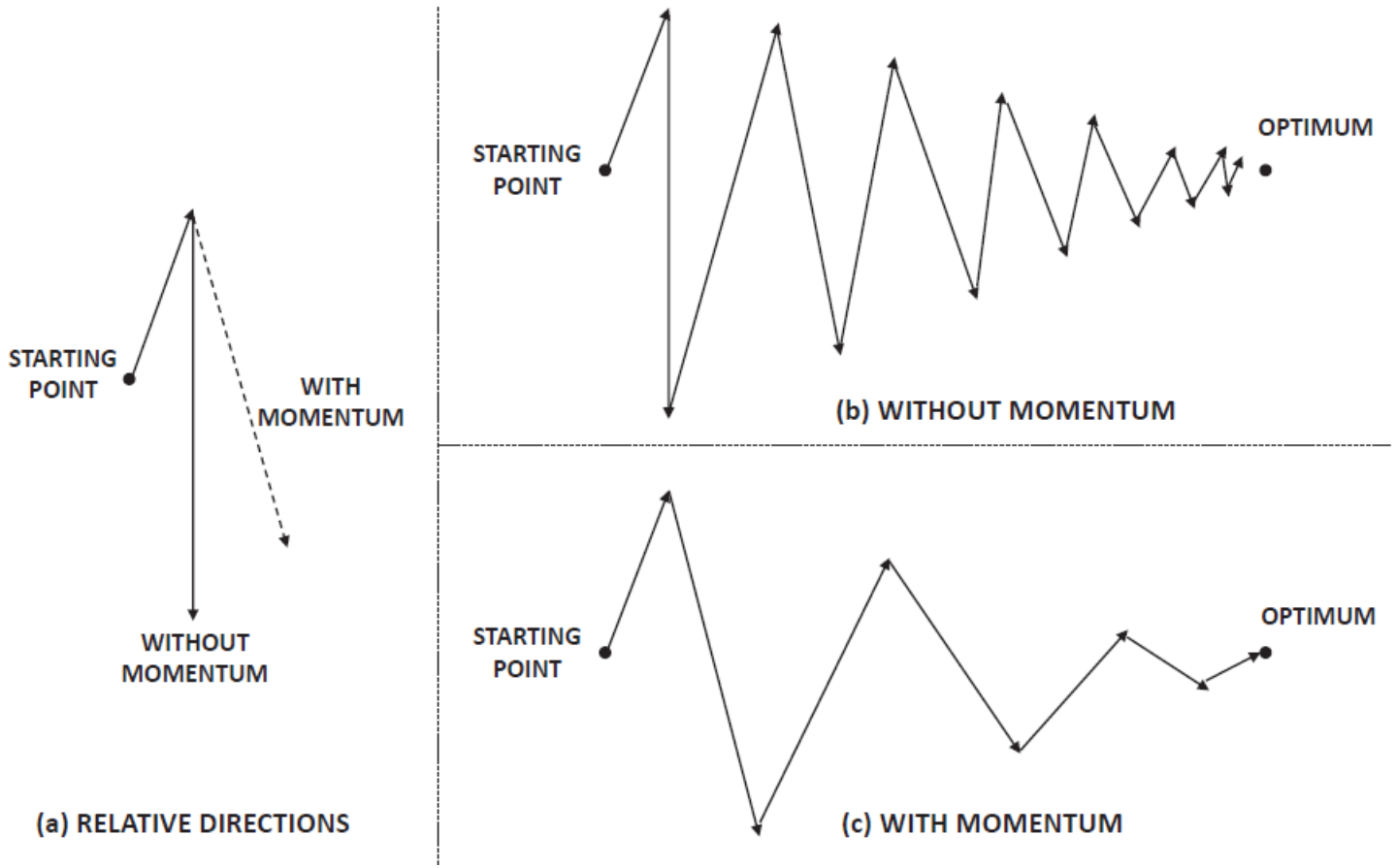
Momentum Methods (cont.)

Main idea:

- Update is accumulated over a number of steps of previous updates
  - Newly computed gradient is only a fraction of total update
  - Give greater preference to *consistent* directions over multiple steps
- Momentum update will often cause the solution to slightly overshoot in the direction where velocity is picked up, just as a marble will overshoot when it is allowed to roll down a bowl.



Avoiding Zig-Zagging with Momentum



- Momentum increases the relative component of the gradient in the correct direction

Nesterov Momentum

- Modification of the traditional momentum method in which the gradients are computed at a point that would be reached after executing a β-discounted version of the previous step again, so over time contribution of long-ago steps is discounted by β^t (exponentially discounted accumulation).
- Compute at a point reached using only the momentum portion of the current update:

$$\vec{v} \Leftarrow \underbrace{\beta\vec{v}}_{\text{Momentum}} - \alpha\frac{\partial L(\vec{w} + \beta\vec{v})}{\partial\vec{w}}$$

and

$$\vec{w} \Leftarrow \vec{w} + \vec{v}$$

where $L(\vec{w})$ the loss function at the current solution $\vec{w}$

Nesterov Momentum - Meaning

- The idea is that this corrected gradient uses a better understanding of how the gradients will change because of the momentum portion of the update, and incorporates this information into the gradient portion of the update.
- We are using a certain amount of lookahead in computing the updates
- Using the value of the gradient a little further along the previous update can lead to faster convergence – in downhill analogy put on the brakes as the marble reaches near bottom of hill.
- Nesterov momentum should always be used with mini-batch SGD (rather than SGD).

Momentum in Keras

- Momentum is still just a correction for SGD so SGD method must be used, for example:

```
opt = tf.keras.optimizers.SGD(learning_rate=xxx,\
        momentum=yyy, nesterov=True)
```

where xxx is learning rate (by default is 01.) momentum is the β value (default is 0.9) and nesterov is Boolean variable to apply nesterov connection or not with default being False.

Lecture Overview

1. Tuning Learning Rate
2. Momentum Methods
3. AdaGrad and RMSProp
4. AdaDelta and Adam

Parameter Specific Learning

- Momentum – the idea of consistency across all parameters
- Observation: parameters with large partial derivatives are often oscillating and zigzagging, whereas parameters with small partial derivatives tend to be more consistent, but move in the same direction.
- The idea make adjustments specific to values of gradient parameters and apply momentum only when necessary
- Must keep track of the magnitude of the partial derivative with respect to each parameter over the course of the algorithm
- To use momentum-like approach need to aggregate it

AdaGrad

- Keeps track of the aggregated squared magnitude (ASM) of the partial derivative with respect to each parameter and normalize update by the parameter
  - Square-root of ASM is *proportional* to the root-mean-square slope for that parameter
    - Initialized to 0, the absolute value will increase with the number of epochs because of successive aggregation:

$$A_i \Leftarrow A_i + \left(\frac{\partial L}{\partial w_i}\right)^2; \quad \forall i$$

  where A_i is aggregate value of i^{th} parameter
- The update of the parameter w_i is the

$$w_i \Leftarrow w_i - \frac{\alpha}{\sqrt{A_i}}\left(\frac{\partial L}{\partial w_i}\right); \quad \forall i$$

- α is the meta-parameter called global learning rate
- Use $\sqrt{A_i + \epsilon}$ in the denominator to avoid ill-conditioning. Here, ϵ is a small positive value such as 10^{-8} which is mandatory because of initialization

AdaGrad Intuition

- Scaling the derivative inversely with $\sqrt{A_i}$ encourages faster *relative* movements along gently sloping directions
  - Wildly fluctuating (zigzagging) components from positive high value to negative high value will be penalized
  - Very active sloping components will be penalizes
- So all components slow down with time because of aggregation

Problem:

- Aggregate scaling factors depend on ancient history, which can eventually become stale
- It might prematurely become too slow, and it will eventually (almost) stop making progress.

AdaGrad in Keras

```
opt = tf.keras.optimizers.Adagrad( \
        learning_rate=0.001, \
        initial_accumulator_value=0.1, \
        epsilon=1e-07, name="Adagrad")
model.compile(loss='mse', optimizer=opt))
```

where

- initial_accumulator_value is the starting value for the accumulators (per-parameter momentum values – it must be non-negative).
- Epsilon - small floating point value used to maintain numerical stability.

RMSProp

- Similar to AdaGrad but aggregation is computed as

$$A_i \Leftarrow \rho A_i + (1-\rho)\left(\frac{\partial L}{\partial w_i}\right)^2; \quad \forall i$$

  where $\rho \in (0,1)$.
- The same update rule as in AdaGrad with (global) learning rate α is used:

$$w_i \Leftarrow w_i - \frac{\alpha}{\sqrt{A_i}}\left(\frac{\partial L}{\partial w_i}\right); \quad \forall i$$

- So we get exponential averaging in normalization constant A_i:
  - The contribution of an epoch that happened t epoch ago will be weighted by ρ^t within current value of A_i contributions of early gradients disappear relatively quickly
  - Absolute magnitudes of scaling factors do not grow with time.
  - Problem of staleness is much smaller.
- Suffers from low normalization values at early stages

RMSProp with Nesterov Momentum

- The same update is computed by computing

$$v_i \Leftarrow \underbrace{\beta v_i}_{\text{Momentum}} - \frac{\alpha}{\sqrt{A_i}} \left(\frac{\partial L(\vec{w} + \beta \vec{v})}{\partial \vec{w}} \right)$$

where $\beta \in (0,1)$ and

$$w_i \Leftarrow w_i + v_i \quad \forall i$$

partial derivative of the loss function is computed at a shifted point $\vec{w} + \beta \vec{v}$, as is common in the Nesterov method.

- Similarly aggregation is computed as the same shifted point

$$A_i \Leftarrow \rho A_i + (1-\rho) \left(\frac{\partial L(\vec{w} + \beta \vec{v})}{\partial w_i} \right)^2 ; \quad \forall i$$

where $\rho \in (0,1)$

- Approach combines advantages but still suffers from low normalization values at early stages

Lecture Overview

1. Tuning Learning Rate
2. Momentum Methods
3. AdaGrad and RMSProp
4. AdaDelta and Adam

AdaDelta and Adam Intuition

- Both methods derive intuition from RMSProp
  - AdaDelta track of an exponentially smoothed value of the *incremental changes* of weights Δw_i in previous iterations to decide parameter-specific learning rate.
  - Adam keeps track of exponentially smoothed gradients from previous iterations (in addition to normalizing like RMSProp).
- Adam is extremely popular method.

AdaDelta

The idea:

In each update, we can obtain the value of Δw_i that is the increment in the value of w_i. If we accumulate it and factor it into update then, if we are zig-zagging on this parameter, the accumulation will reflect that and make updates much smaller

- Compute accumulation δ_i of exponentially smoothed values of Δw_i with the same decay parameter ρ as we computed the accumulation of gradient A_i in RMSprop:

$$\delta_i \Leftarrow \rho\delta_i + (1 - \rho)(\Delta w_i)^2; \forall i$$

- The parameter update is then

$$w_i \Leftarrow w_i - \underbrace{\sqrt{\frac{\delta_i}{A_i}}\left(\frac{\partial L}{\partial w_i}\right)}_{\Delta w_i}; \quad \forall i$$

where as in RMSProp $A_i \Leftarrow \rho A_i + (1-\rho)\left(\frac{\partial L}{\partial w_i}\right)^2$

AdaDelta (cont.)

- There is the difference between computation of $\delta_i \Leftarrow \rho\delta_i + (1 - \rho)(\Delta w_i)^2$ and computation of $A_i \Leftarrow \rho A_i + (1-\rho)\left(\frac{\partial L}{\partial w_i}\right)^2$ in RMSProp:
  - For a given iteration, the value of δ_i can be computed using only the iterations before it because the value of Δw_i is not yet available
  - For a given iteration A_i can be computed using the partial derivative in the current iteration!
- Eliminates the need for a global learning parameter α by computing it as a function of incremental updates in previous iterations. It mitigates zigzagging by making δ_i very small whenever the parameter oscillates.

ADAM

The idea.

- In RMSprop we are exponentially smoothing accumulation of squared gradient for use in normalization constant. However gradient itself is normalized and accumulated as is. So in smoothing we account for magnitude but not the sign. What if recently we are moving in the right direction with high magnitude? shouldn't we continue and not slow down by high magnitude? This can be achieved by exponential smoothing of gradient itself.
- If we are zigzagging then smoothing of gradient itself will only diminish updates as in momentum methods - not increase them
- So let's apply square exponential smoothing as normalization constant and also smooth the gradient itself for updates, i.e. combine momentum and normalization ideas

ADAM Formally

- Normalization constant is updated as in RMSProp with decay parameter $\rho_A \in (0,1)$:
- $A_i \Leftarrow \rho_A A_i + (1-\rho_A)\left(\frac{\partial L}{\partial w_i}\right)^2; \quad \forall i$
- Gradient speed is updated as in momentum method but instead of global parameters $\alpha, \beta \in (0,1)$ we are going to use single parameter ρ_v (not the same as in normalization constant):
- $v_i = \rho_v v_i - (1-\rho_v)\left(\frac{\partial L}{\partial w_i}\right); \quad \forall i$
- Then the parameter update on t^{th} iteration is done as:

$$w_i \Leftarrow w_i - \frac{\alpha_t}{\sqrt{A_i}} v_i; \quad \forall i$$

where $\alpha_t = \alpha\left(\frac{\sqrt{1-(\rho_A)^t}}{1-(\rho_v)^t}\right)$

- Here ρ_A, ρ_v, α are global parameters

More on Parameters of ADAM

- Both v_i and A_i are initialized to 0 which causes total bias to large upgrades in early iterations. But these quantities are affected differently as one is linear in its global parameter the other is square-rooted in it global parameter, so this bias is adjusted in the formula for α^t
- each of $(\rho_A)^t$ and $(\rho_v)^t$ converge to 0 as t diverges because $\rho_A, \rho_v \in (0,1)$. As a result, the initialization bias correction factor $\frac{\sqrt{1-(\rho_A)^t}}{1-(\rho_v)^t}$ converges to 1, and α_t converges to α.
- The default suggested values of ρ_v and ρ_A are 0.9 and 0.999, respectively, according to the original Adam paper
- Just like other methods ADAM uses $\sqrt{A_i + \epsilon}$ in the denominator instead of A_i for better conditioning

Reading

- Chapter 3.5.3